CS42/413

Introduction to Compilers
Radu Rugina

Lecture 30: Instruction Selection
14 Apr 04

Instruction Selection

- Different sets of instructions in low-level IR and in the target machine
- Instruction selection = translate low-level IR to assembly instructions on the target machine
- Straightforward solution: translate each low-level IR instruction to a sequence of machine instructions
- Example:
  \[
  x = y + z \quad \text{\texttt{mov} } y, r1 \\
  \text{mov } z, r2 \\
  \text{add } r2, r1 \\
  \text{mov r1, x}
  \]

Backend Optimizations

- Instruction selection
  - translate low-level IR to assembly instructions
  - A machine instruction may model multiple IR instructions
  - Especially applicable to CISC architectures
- Register Allocation
  - Place variables into registers
  - Avoid spilling variables on stack

Example

- Consider the computation: \( a[i+1] = b[j] \)
- Assume \( a, b, i, j \) are global variables
  - register ra holds address of a
  - register ri holds value of i
  - register rj holds value of j

Low-level IR:

\[
\begin{align*}
  t1 &= j^4 \\
  t2 &= b+i1 \\
  t3 &= i^2 \\
  t4 &= i+1 \\
  t5 &= t4^4 \\
  t6 &= a+t5 \\
  ^*t6 &= t4
\end{align*}
\]

Possible Translation

- Address of \( b[j] \):
  \[
  \text{mulc } 4, rj \\
  \text{add } rj, rb \\
  ^*t3 = t2 \\
  t4 = i+1 \\
  t5 = t4^4 \\
  ^*t6 = t4
  \]
- Load value \( b[j] \):
  \[
  \text{load } rb, r1 \\
  \text{add r1, ra}
  \]
- Address of \( a[i+1] \):
  \[
  \text{add r1, ra} \\
  \text{mulc } 4, ri \\
  \text{add } ri, ra \\
  \]
- Store into \( a[i+1] \):
  \[
  \text{store } r1, ra
  \]
Another Translation

- Address of b[j]: `mulc 4, rj
add rj, rb`

- Address of a[i+1]: `add 1, ri
mulc 4, ri
add ri, ra`

- Store into a[i+1]: `movem rb, ra`

Low-level IR:

```
t1 = j*4
t2 = b+1
r3 = t2
t4 = i+1
t5 = t4*4
t6 = a+t5
*t6 = t4
```
Tiling

• Tiling = cover the tree with disjoint tiles

Assembly:
movn rb, ra
store
movn rb, ra
load
add rj, rb
mulc 4, rj
add 1, rl
mulc 4, rl
add ri, ra

Directed Acyclic Graphs

• Tree representation: appropriate for instruction selection
  – Trees = subtrees -> machine instructions

• DAG = more general structure for representing instructions
  – Common sub-expressions represented by the same node
  – Tile the expression DAG

• Example:
t = y + i
y = z * t
z = t * y

Big Picture

• What the compiler has to do:
  1. Translate low-level IR code into DAG representation
  2. Then find a good tiling of the DAG
     - Maximal munch algorithm
     - Dynamic programming algorithm

DAG Construction

• Input: a sequence of low IR instructions in a basic block
• Output: an expression DAG for the block

• Idea:
  – Label each DAG node with variable which holds that value
  – Build DAG bottom-up

• A variable may have multiple values in a block
• Use different variable indices for different values of the variable; t, t, t, etc.

Algorithm

index[v] = 0 for each variable v
For each instruction i (in the order they appear)
  For each v that i directly uses, with n=index[v]
  if node v doesn’t exist
  create node v, with label v
  Create expression node for instruction i, with children
  \{ v, v ∈ use[1] \}
  For each v ∈ def[1]
  index[v] = index[v] + 1
  If i is of the form x = ... and n = index[x]
  label the new node with x
Issues

- Function/method calls
  - May update global variables or object fields
  - def[I] = set of globals/fields
- Store instructions
  - May update any variable
  - If stack addresses are not taken (e.g. Java),
  - def[I] = set of heap objects

Next: DAG Tiling

- Goal: find a good covering of DAG with tiles
- Problem: need to know what variables are in registers
- Assume abstract assembly:
  - Machine with infinite number of registers
  - Temporary/local variables stored in registers
  - Parameters/heap variables: use memory accesses

Problems

- Classes of registers
  - Registers may have specific purposes
    - Example: Pentium multiply instruction
      - multiply register eax by contents of another register
      - store result in eax (low 32 bits) and edx (high 32 bits)
      - need extra instructions to move values into eax
- Two-address machine instructions
  - Three-address low-level code
  - Need multiple machine instructions for a single tile
- CISC versus RISC
  - Complex instruction sets => many possible tiles and tilings
  - Example: multiple addressing modes (CISC) versus load/store architectures (RISC)

Pentium ISA

- Pentium: two-address CISC ISA
- Multiple addressing modes: source operands may be
  - Immediate value: imm
  - Register: reg
  - Indirect address: [reg], [imm], [reg+imm],
  - Indexed address: [reg+reg'], [reg+imm+reg'],
  - [reg+imm*reg'+imm]
- Destination operands = same, except immediate values

Example Tiling

- Consider: \( t = t + i \)
  - \( t \) = temporary variable
  - \( i \) = parameter
- Need new temporary registers between tiles (unless operand node is labeled with temporary)
- Result code:
  - \texttt{mov} \%ebp, t0
  - \texttt{sub} $20, t0
  - \texttt{mov} 0(t0), t1
  - add t1, t
- Note: also compute \( i \), if it is live

Some Tiles

- Move \( t2, t1 \)
- Store \( t1, t2 \)
- \texttt{mov} \( t1 \), \%eax
  - \texttt{mul} \( t2 \)
  - \texttt{mov} \%eax, \( t3 \)
Conditional Branches

- How to tile a conditional jump?
- Fold comparison into tile

```
t1
test t1,t1
jnz L

L
```

```
t1
t1
cmp t1,t2
je L
```

Load Effective Address

- Lea instruction computes a memory address
- Doesn't actually load the value from memory

```
t1
L
lea (t1,t2), t3
```

```
t2
L
lea (t1,t2,8), t3
```